AR-006-482

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

Research Report
ERL-0545-RR

REPORT ON m-EVES

by

Jim Grundy

SUMMARY

This report reviews the verification tool m-EVES (Environment for Verifying and Evaluating Software). It describes the two main components of m-EVES: the language, m-Verdi and the theorem prover, m-NEVER. The main purpose of this report is to describe how problems are attacked in m-EVES, and draw conclusions about the strengths and weaknesses of m-Verdi and m-NEVER. Where appropriate, comparisons are made with other program verification and theorem proving environments.

MAR 91

APPROVED FOR PUBLIC RELEASE

POSTAL ADDRESS: Director, Electronics Research Laboratory,PO Box 1600, Salisbury, South Australia, 5108.

Contents

1 INTRODUCTION 1
2 m-VERDI 1
2.1 Declarations 2
2.1.1 Type Declarations 2
2.1.2 Constant Declarations 2
2.1.3 Variable Declarations 3
2.1.4 Variable Bindings 3
2.1.5 Function Declarations 4
2.1.6 Procedure Declarations 4
2.2 Commands 5
2.2.1 abort 5
2.2.2 exit 5
2.2.3 return 6
2.2.4 Annotation 6
2.2.5 Assignment 6
2.2.6 Conditional 6
2.2.7 Iterative 7
2.2.8 Procedure Invocation 7
2.2.9 block 7
2.3 Expressions 8
2.3.1 Quantification 8
2.3.2 Structured Results 8
2.3.3 Conditional Expressions 8
2.4 Machine Support 8
2.5 Conclusions 9
2.5.1 Strengths of m-Verdi 9
2.5.2 Weaknesses of m-Verdi 10
3 m-NEVER 12
3.1 Declarations 12
3.1.1 Facts 12
3.1.2 Rewrite Rules 12
3.1.3 Forward Rules 14

3.2 Commands 16
3.2.1 Simplification 17
3.2.2 Rewriting 17
3.2.3 Substitution 17
3.2.4 Reduction 18
3.2.5 Definitions 19
3.2.6 Case Analysis 19
3.2.7 Induction 19
3.2.8 Normal Forms 20
3.2.9 Quantifiers 20
3.2.10 Counter Examples 20
3.2.11 State Manipulation 20
3.3.1 Strengths of m-NEVER 21
3.3.2 Weaknesses of m-NEVER 21
4 DOCUMENTATION 25
5 CONCLUSIONS 27
6 ACKNOWLEDGMENTS 28
REFERENCES 29
APPENDIX A: BINARY SEARCH PROBLEM 30

1 INTRODUCTION

m-EVES (Environment for Verifying and Evaluating Software) is an environment for the development of formally verified software. m-EVES has two main components: a language, m-Verdi, and a theorem prover, m-NEVER (Not the EVEs Rewriter). The purpose of this report is to introduce and give some insight into m-Verdi and m-NEVER, explain how to use them, and draw some conclusions about their strengths and weaknesses. Where appropriate, comparisons have been drawn with similar features from some other tools.

m-Verdi is a combined specification and programming language. m-Verdi is similar in style and expressive ability to Pascal, but with extensions which allow specification and verification by including assertions, pre and post conditions, and invariants and termination conditions on loops. The specification facilities of m-Verdi are essentially those of first order predicate calculus.

The m-NEVER theorem prover is a powerful theorem proving tool and in many ways is similar to the Boyer-Moore [1] theorem prover. Perhaps the most significant difference between m-NEVER and Boyer-Moore is that m-NEVER allows a degree of user interaction to guide the proof as it progresses, while Boyer-Moore is batch oriented.

A typical session with m-EVES is commenced using the ECL (EVES Command Language) to make declarations in m-Verdi of various items such as types, variables, procedures or functions. The user may also make declarations of axioms and inference rules for use by m-NEVER. These declarations give rise to various proof obligations which the user proves with m-NEVER by issuing commands in ECL. Although m-Verdi and m-NEVER are two distinct components, a user views the system as a unified whole, with ECL the common interface.

The following font convention is observed throughout this document. Normal text is set in roman. Examples drawn from m-EVES are set in `typewriter`. Examples drawn from other languages and systems are set in sans-serif. m-Verdi key words, and keywords drawn from other language examples will be emboldened, that is in **`bold typewriter`** and **bold sans-serif** respectively. In any sample interactions with any system that portion of the interaction attributed to the user will be underlined. In examples, words in italics denote generic values, for example, the word "name" in "`"`*`name`*`"`" stands for any name.

2 m-VERDI

The version of m-Verdi used by the m-EVES system is a subset of the language as described in the papers *A Description of m-Verdi* [2] and *An m-Verdi User's Guide* [5]. For example, packages and environments are not yet supported by m-EVES. Unless otherwise stated, this report will use "m-Verdi" to mean the language supported by m-EVES version 4.

The m-Verdi language was designed specifically for use with the m-EVES software verification effort. Unlike other languages designed for use in verification projects, like MALPAS Intermediate Language (IL) [15], m-Verdi is intended as a programming language in its own right rather than an intermediate language into which programs in existing languages are translated. There is a compiler available for m-Verdi (the complete version), although it is not part of m-EVES.

Examination of m-Verdi suggests that it was designed to have a simple semantics, which is easily formally manipulated, while maintaining a high degree of expressiveness. In instances where the m-Verdi language seems unusual it is invariably the case that the semantics of the language construct in question is demonstrably simpler than that of the more traditional construct.

The first thing that a new user of m-Verdi will notice is the almost exclusive use of prefix notation in expressions, and the use of words rather than symbols to denote common mathematical functions. For example the traditional Pascal expression: `1 - 2` is rendered as `minus(1, 2)` in m-Verdi. The only exceptions to this rule are the "equals" and "not-equals" functions: `=` and `<>`.

Second, a new user of m-Verdi will notice the almost total lack of polymorphic functions. Users of traditional procedural programming languages, while not being able to write polymorphic functions of their own, are familiar with their use; often without realising it. For example, the addition function in Pascal works for both integers and real numbers. Even this limited degree of polymorphism is not supported in m-Verdi. m-Verdi has two numeric types, ordinal and integer, each of which has its own less-or-equal function, namely: `ordinal'le` and `int'le`. Again the only exceptions to this rule are the equals and not-equals functions. Perhaps the most striking example of these two effects combined is in array indexing. In a language like Pascal, array indexing could be considered to be the work of an infix polymorphic function _ [_] which takes two arguments, an array and an index. In m-Verdi the Pascal expression: a[i] is written as `atype'extract(a, i)`, where `atype` is the type of the array `a`, and `atype'extract` is the indexing function for arrays of type `atype`.

2.1 Declarations

2.1.1 Type Declarations

m-Verdi supports the following standard types: integer (`int`), boolean (`bool`), ordinal (`ordinal`) and character (`char`). m-Verdi allows the declaration of array types, record types, enumeration types and a generalised version of subrange types called *restriction types*. Defining a type `t` also results in the declaration of the variable `t'dummy` of that type.

By using a restriction type declaration, a new type `t` may be defined as a restriction of a larger type, *T*, by giving a predicate on *T* which defines the subset `t`. For example, the natural numbers could be defined as a restriction on the integers such that the naturals are those integers greater than or equal to zero. Restriction types have a proof obligation that the type declared is nonempty. Making use of restriction types in procedure declarations complicates their proof obligations as the user is required to show that variables of a restriction type satisfy the type's defining predicate. It is often the case that the practical disadvantages of using restriction types outweigh the advantages inherent in their conceptual beauty. Below is the restriction type declaration of the natural numbers:

```
type Nat = those int'dummy : int'le(0, int'dummy)
```

Certain types, namely ordinals and restriction types, are *nonexecutable types*. Any construct making use of a nonexecutable type will be nonexecutable. Nonexecutable constructs will not be compiled by the m-Verdi compiler.

Array declarations give rise to a proof obligation that the array declared is not empty. After entering an array declaration in m-EVES, the proof obligation that the array is not empty will be displayed, and the user has the opportunity to use m-NEVER to prove the obligation.

Types may also be declared which are not associated with any particular structure. Such declarations are called *type stubs*. It should be noted that once a type has been declared as a stub it cannot later be refined to a particular structure. Below is an example of a type stub declaration:

```
type atype
```

2.1.2 Constant Declarations

Typed constants may be defined. These constants may or may not have a value associated with them. Constant declarations which do not associate a value with the constant are called *constant*

stubs. Constant stubs like type stubs may not later be refined to have a particular value. The following statements define two constants; `three`, an integer constant equal to 3, and `anint`, an integer constant with no defined value:

```
const three : Int = plus(1, 2)
const anint : Int
```

2.1.3 Variable Declarations

The concept of a variable declaration in m-Verdi is markedly different from that in traditional programming languages. A variable declaration in m-Verdi does not create a program variable with an associated state; indeed, there are no global variables in m-Verdi. A variable declaration in m-Verdi adds the variable symbol to the theory. Effectively, a variable declaration associates a type with a variable name. The task traditionally associated with variable declarations is done by *variable bindings*. The distinction between declaring a variable and allocating storage for it is made because there are a great many uses for variables in m-Verdi, including specifications and axioms, which require no physical storage.

Another difference between m-Verdi and more traditional languages is that a variable declaration declares not just one variable, but a family of variables. For example, declaring the variable v effectively declares the variable family: v, $v'0$, $v'1$, $v'2$, ... Below the variable `b` is declared as a boolean:

```
var b : Bool
```

2.1.4 Variable Bindings

Once declared, a variable may be used in nonexecutable contexts, like specifications and axiom declarations. However, before being used in an executable portion of an m-Verdi program, a variable must be given a binding. A variable may be bound in one of three modes: **`lvar`**, **`pvar`** and **`mvar`**.

An **`lvar`** or *logical variable* binding associates an immutable value with a variable for the duration of its scope. In this respect, it is much like a constant declaration in a traditional programming language.

A **`pvar`** or *program variable* binding is much like a traditional variable declaration. The variable denotes its value in the current state and assignment statement may be used to change that value.

An **`mvar`** or *machine variable* binding form part of m-Verdi's support for modelling the architecture of the target machine. Machine variables are used for modelling special registers which do not behave like ordinary variables. For example, it is common for communications chips to contain a register which places the value stored in it on the output port, but when read returns the value present on the input port. A machine variable can only be modified or examined by procedures which the user specifies so as to model the behaviour of the physical register.

A variable may be given a binding either in the definition of a procedure's formal parameters, or at the start of a block.

Within a procedure, **`lvar`** bound parameters behave like constants which have their values passed to them by the corresponding actual parameters; **`pvar`** bound parameters have a call by reference semantics, denoting the corresponding actual parameter. To avoid the complications of call by reference semantics, aliasing is not permitted in m-Verdi procedure calls.

Bindings at the beginning of a block may include an optional initialisation. Variables given **lvar** and **pvar** bindings at the beginning of a block behave, respectively, like traditional constants and variables within that block.

Examples of bindings may be found in sections 2.1.6 and 2.2.9 on procedure declarations and blocks.

2.1.5 Function Declarations

Functions in m-Verdi are essentially named, parameterised expressions, much like the functions of a functional programming language. There is no equivalent to the Pascal style of function which is like a procedure that returns a value.

A function need not have a body; this kind of function declaration is known as a *function stub*. A function stub declaration only requires the function's name, its formal parameters and the type which it returns. Function stubs are useful for developing and reasoning about algebraic theories. For example, a user may declare type stubs `elt` and `eltlist` for element and element list; and function stubs `cons` and `head` to prepend an element to a list and to take the head of a list respectively. The user may then assert that taking the head of a list that has just had an element prepended to it returns that element. In such a way abstract theories of lists and sets and the like may be built up with m-Verdi and m-NEVER. Like other forms of stub, function stubs may not be given definitions later in the program.

Functions may be recursive, but not mutually recursive. Recursive functions must contain an expression, called the *measure*, which represents the size of the problem to be solved. Generally, termination may be established by showing that the measure decreases, according to some well founded ordering, on successive evaluations. However, in m-Verdi the well founded ordering is always less-than on the ordinal numbers. This restriction detracts from the expressiveness of the language as the size of many problems is not most naturally expressed in terms of the ordinals. The difficulty of showing termination of mutually recursive functions was probably considered too great to justify their inclusion in the language. Recursively defined functions are not executable, and are merely used for specification purposes. The stack usage of such functions would make it very difficult to check their storage usage.

Below is an example of a function declaration for calculating the factorial of an integer, `i`.

```
var i : Int
function factorial(i) : Int =
pre int'ge(i, 0)
measure ordinal'val(i)
begin
  if i = 0 then
    1
  else
    times(i, factorial(minus(i, 1)))
  end if
end factorial
```

2.1.6 Procedure Declarations

Procedure declarations in m-Verdi are similar to those in Pascal, the main differences being that pre and post-conditions given in predicate logic may be included in an m-Verdi procedure definition.

m-Verdi procedure definitions may also include an *initial* clause to initialise various logical variables to values in the state before the procedure executes. Unlike functions, m-Verdi procedures may not be recursively defined.

Procedure Stubs are defined by omitting the block containing the procedure's code. m-EVES can still reason about software containing procedure stubs by using the pre and post conditions of the stubs. Procedure stubs allow users to suppose they have a set of procedures which perform certain functions, then write and verify software based around those procedures, without needing to first code and verify the procedures. Procedure stubs may not be given their defining code later in the same program.

Below is an example of a definition of a procedure to search for a `target` within an array `a`.

```
procedure search(lvar a, lvar target, pvar position) =
pre some i : and(int'ge(i, atype'lowbound),
                 int'ge(atype'highbound, i),
                 atype'extract(a, i) = target)
post and(int'ge(position, atype'lowbound),
         int'ge(atype'highbound, position),
         atype'extract(a, position) = target)
begin
   ⋮
end search
```

2.2 Commands

The m-Verdi language offers a range of commands similar to that found in most small procedural languages.

2.2.1 abort

The **abort** command is used for terminating program execution. An `abort` command may have a condition. An example of **abort** follows:

```
abort when int'lt(i, 0)
```

2.2.2 exit

The **exit** command, with its optional condition, is used to exit loops in a manner similar to C's break command. A usage of **exit** appears as an example below:

```
exit when i = 0
```

2.2.3 return

The **return** command is used to return prematurely from a procedure invocation. The syntax of the return command is the same as the **exit** and **abort** commands, including the optional condition. A typical usage of **return** looks like the one below:

```
return when i = 0
```

2.2.4 Annotation

Annotations are used to make assertions at various points throughout a procedure. Annotations affect a procedure's proof obligation. The user is required to prove that the precondition will imply the correctness of the annotation, and that together they will imply the postcondition. Annotations are made with the **note** command, an example of which follows:

```
  note gcd(i, j) = gcd(i'0, j'0)
```

2.2.5 Assignment

m-Verdi assignments are much like those of Pascal, the only exception being the assignment of values to elements of structured variables. In languages like Pascal individual elements within structured variables may be assigned values directly using the same notation to identify the element as is used to extract its value in an expression. In m-Verdi values are *immutable*. They may only be assigned to whole variables; to update a particular element of a structured variable one assigns to the variable the result of some function which returns the variable updated in the way desired. As an example, the m-Verdi equivalent of the Pascal a[i] := e is shown below:

```
a := atype'with(a, i, e)
```

Users of m-Verdi are free to use the more traditional notation: `a(i) := e`. However, this notation is regarded as a shorthand and is immediately translated into the form shown above.

2.2.6 Conditional

Conditional statements in m-Verdi are similar to those in other procedural programming languages. The example below shows the usage of m-Verdi's conditional statement:

```
if int'gt(i, j) then
  g := gcd(minus(i, j), j)
elseif i = j then
  g := i
else
  g := gcd(i, minus(j, i))
end if
```

2.2.7 Iterative

The iterative construct of m-Verdi differs significantly from the iterative constructs of similar procedural languages. While many languages support several different kinds of loops, commonly **while**, **repeat** and **for**, m-Verdi supports only one. The m-Verdi **loop** command is basically an infinite loop. The only way to leave the loop is with an **exit** command.

m-Verdi loops are augmented with invariants to help prove correctness. m-Verdi loops, like recursive functions, also may be augmented with measures to establish termination. Loops without explicit measure conditions are given the constant expression `ordinal'first` (the smallest ordinal number) as a measure. This constant expression cannot be shown to decrease, and therefore such loops cannot be proven correct.

The loop from a procedure for calculating the greatest common divisor of two numbers is given below as an example (note that `ordinal'val`, which appears in the measure, is a type casting function from nonnegative integers to ordinals):

```
loop
  invariant and(int'gt(i, 0),
                int'gt(j, 0),
                gcd(i, j) = gcd(i'0, j'0))
  measure ordinal'val(plus(i, j))
  exit when i = j

  if int'lt(i, j) then
    j := minus(j, i)
  else
    i := minus(i, j)
  end if
end loop
```

2.2.8 Procedure Invocation

Procedure invocation uses the standard notation. As an example, an invocation of the procedure `search` defined above is given below:

```
search(a, 1, i)
```

2.2.9 block

The **block** command is used for creating a local scope into which variables may be introduced and a sequence of commands executed, similar to the compound statement in C. An example of a block is shown below:

```
block
  lvar i := 1
  lvar j := 2
  pvar k

  k := 0
  k := plus(plus(k, i), j)
end block
```

2.3 Expressions

Expressions in m-Verdi are too varied to be treated in a case-by-case manner as was done with commands. This section examines only differences between m-Verdi expressions and expressions of traditional procedural programming languages like Pascal. The most obvious difference is in syntactic style, but there are some differences which run deeper than appearances.

2.3.1 Quantification

Existential and universal quantification are available as m-Verdi expressions. Such expressions are not executable and may only usefully appear within pre and post-conditions, and invariants. The quantifiers conform to the following syntactic template, where $P[v_1, v_2, ...]$ is a boolean valued expression on $v_1, v_2, ...$:

some $v_1, v_2, ... : P[v_1, v_2, ...]$
all $v_1, v_2, ... : P[v_1, v_2, ...]$

2.3.2 Structured Results

Unlike Pascal and many other procedural languages, functions in m-Verdi may return structured values such as arrays and records as their results.

2.3.3 Conditional Expressions

m-Verdi also allows for conditional expressions which have a similar syntax to the conditional command described above. The factorial function declared previously contained the following conditional expression:

```
if i = 0 then
  1
else
  times(i, factorial(minus(i, 1)))
end if
```

2.4 Machine Support

Critical software is frequently found running on small embedded systems where its operation is intimately intertwined with the hardware. This section describes m-Verdi's support for modelling the way in which programs will run on their target hardware. Concepts introduced are avoided in other parts of the report as it is challenging enough to prove a nontrivial algorithm correct without proving that it will run correctly on your chosen hardware.

Registers on micro chips, while essentially being variables, cannot be manipulated in the same way as ordinary variables, and should not be modelled as such. For example, it is common for a register to reset its value to zero after it has been read. m-Verdi includes a facility for modelling registers called *machine variables*. A machine variable can only be manipulated by procedures which the user declares to model the register's behavior.

m-Verdi includes a facility for modelling imperfect machine arithmetic. For example the addition function on integers commonly used in mathematics (the *ideal* version of the function) is not the same as the addition function implemented in programming languages (a *non-ideal* approximation to the ideal function). Consider a machine which represents integers by 2s complement 16 bit words (possibly the most common representation used by the small computers found in embedded systems). On such a machine integers in the range -32768 to $+32767$ may be represented. The addition function in a programming language is only defined if its arguments and their sum are representable integers. m-Verdi handles this by making a distinction between *executable functions* and *non-executable functions*. Executable functions are those functions which are to run on the machine. Non-executable functions are ideal functions which either have no executable version or for which the corresponding executable versions are non-ideal. Non-executable functions are intended for use in specification. For example, one could specify a procedure to add two numbers using the executable addition function and prove that so long as the arguments fell within a certain range the procedure adds the numbers correctly (according to the ideal function). Non-executable functions may also be used within procedures. However, such a procedure will then be deemed *non-executable*. For the purposes of this report only ideal functions, whether executable or non-executable, will be used.

Although m-Verdi is capable of modelling machine registers and machine arithmetic, it has not attempted to address the remaining question of storage exhaustion. It is not possible to place restrictions on the size or number of variables allocated or to put an upper bound on the number of procedure and function calls active at any one time, and hence on the runtime stack size. This means that in order to ensure that a program will run correctly it is necessary to manually check that storage allocation cannot exceed the target hardware's memory capacity.

2.5 Conclusions

2.5.1 Strengths of m-Verdi

Although poor in expressiveness when compared to languages like Z [16] and VDM [8], m-Verdi's specification ability is relatively good when contrasted with a language such as MALPAS IL which does not support quantification.

The simple semantics of m-Verdi means that programs written in m-Verdi have relatively simple verification conditions and hence are ideal for use in a verification environment. This simple semantics and its formal definition are the major strengths of m-Verdi — particularly when compared with languages like MALPAS IL which have no formally defined semantics. A simple program semantics also makes a programming language easy for people to reason about, thereby increasing the probability that programs written in the language will be correct in the first place.

Despite its simple semantics m-Verdi does have some programming facilities not commonly found in procedural programming languages. In particular m-Verdi functions may return structured results like arrays and records and m-Verdi allows the use of conditional i.e. **`if _ then _ else _ end if`** expressions.

m-Verdi's inclusion of measures in the definition of recursive functions and loops allows proofs of program termination, and hence total correctness, to be constructed. Some existing systems, such as MALPAS and Gypsy [6], do not provide direct support for proofs of total correctness.

m-Verdi's variable family concept helps theorem provers to generate new variable names which show their relationship to existing variables. For example, if a predicate contains a quantified variable `i` and m-NEVER needs to create a second instance of the expression containing `i` it can call the variable equivalent to `i` in the new predicate `i'1`. Some systems which automatically generate variables give them names like genvar1234, which mean little to the user.

The inclusion of various facilities in m-Verdi for modelling a program's target hardware allows, in theory at least, proofs of program correctness which not only show that the program's algorithm is correct, but that, excluding the possibility of memory exhaustion, it will execute correctly on its target hardware. This is an important question not directly addressed by other verification tools. In particular the common assumption that machine arithmetic is ideal makes it difficult for some verification systems to discover overflow errors, which can have catastrophic effects.

2.5.2 Weaknesses of m-Verdi

The most obvious weakness of m-Verdi is its visual appearance. Although the look and feel of the language may not seem, at first, of fundamental significance to the value of the language within a verification context, it is surprisingly relevant. Use of m-EVES requires the ability to read and understand predicates. Typically, a user would scan the predicate being manipulated, issue a proof command and then look at the resulting predicate to ascertain what changes have been made. If the language in which the predicates are expressed is not easy to read then this process is made all the more difficult.

Successful use of m-EVES is greatly assisted by a degree of experience with formal systems and procedural programming. People with such experience will feel more at home with a system which uses a notation with which they are familiar. For example, mathematicians used to working with a notation which allows predicates to be expressed in a form like:

$$\forall i \cdot i + 4 = 5 \Rightarrow i = 1$$

will prefer not to express themselves in m-Verdi, where the expression above would be rendered:

```
all i : implies(plus(i, 4) = 5, i = 1)
```

Although a Lisp-like notation, as used by m-Verdi, is familiar to some programmers, those with a purely procedural background will find m-Verdi aesthetically unappealing.

However, the main problem with m-Verdi is its verbosity. The example above is rather kind to m-Verdi in this respect. Typically an expression when written in m-Verdi is more than twice as big as it would appear in more traditional notation. The difficulty of reading and understanding an expression is greatly increased once the expression spans more than one physical page. When using m-EVES to manipulate large predicates it is usual to obtain a hard copy of the original predicate, perform a planned series of manipulations, get a hard copy of the resulting predicate and compare the two — a process which can take quite some time. The verbosity of m-Verdi has a great effect on the way a user interacts with m-EVES. Typically much time and effort is devoted to ensuring that the predicate being manipulated remains small enough to fit on a page — any manipulations which result in a large increase in the size of the predicate being discarded as too difficult to work with.

There is another problem with m-Verdi of a more serious nature than its appearance. The seemingly intentional simplicity of the language has resulted in too significant a loss in expressiveness to be ignored. In particular the lack of higher order notation and polymorphic typing limit the expressiveness and generality of the language.

As a language for use in a verification context m-Verdi must be suitable for both programming and specification. The purpose of specification is to express the problem in as abstract a manner as possible in order to avoid complexity and hence confusion in the specification. One of the major tools for use against complexity in specifications is higher order notation. m-Verdi does not support higher order notation. Higher order notation is not impossible for theorem proving systems to work with: both HOL [7] and Gypsy allow its use.

m-Verdi does not support polymorphism. This means that it is not possible to use m-EVES to develop general theories. For example, a user cannot develop a general theory of lists. Rather, several theories must be developed, one for each type of list. The lack of polymorphism results in the frequent duplication of both human and machine proving effort. Furthermore, an inability to develop theories of a general nature is an effective block to the development of libraries of useful theories which could increase the productivity of the tool.

Generality ought to be an important concept in software verification. At present full verification is considered too expensive for use in all but the most safety critical applications. This is frequently attributed solely to the difficulty of using theorem provers to prove software correct. Another contributing factor is that languages like m-Verdi do not allow the verification of programs which solve a generic class of problems. Consider a typical software project. As part of the project's design, a module for handling lists of customer orders has been identified. Within this module there is to be a procedure to search the list and return all orders for a particular product. Verification of such a module is clearly uneconomic. However if the design team were to think in terms of a module for handling lists of anything, and in particular a procedure within that module which searches the list, returning all those elements which satisfy a given predicate, then they would have a module so general that it could used in almost all of their company's software projects. The solution to the general problem is no more complex than the solution to the original, more specific, problem. It probably makes good economic sense to verify a module implementing the generic design because of its potential for reuse. m-Verdi could not be used to implement the generic solution. In order to allow m-Verdi procedures to manipulate lists of anything m-Verdi ought to have polymorphic typing. In order for a procedure in m-Verdi to pick out those elements of a list satisfying a given predicate m-Verdi would need to allow functions as first class objects; thus m-Verdi also ought to have higher order notation.

It seems odd that recursive procedures are not allowed in m-Verdi. It could be argued that the inclusion of recursive procedures makes it difficult to reason about the storage consumption of a program. However, this could be accommodated by treating recursive procedures as non-executable, like recursive functions. There are many cases when a user would like to use m-EVES to reason about an algorithm without caring if that algorithm will execute correctly on a machine with finite resources. Disallowing recursive procedures arbitrarily restricts the class of problems to which m-EVES may be applied. Other systems, like MALPAS and Gypsy, do not impose this restriction.

The inability to refine stub declarations is both an inconvenience and a hindrance to the development of a useful system of libraries. It should be possible, for example, to define algebraically, using type and function stubs and axioms, a theory of sets; and later provide a model for that theory by instantiating the type and function stubs with definitions and proving the axioms. In this way the example set theory could potentially be modelled as sets represented by arrays or lists or by some other means. A library of theories could then be developed, each with several models. MALPAS provides an explicit mechanism for refining abstract types, which are its equivalent to type stubs.

One final problem with m-Verdi is simply that it is m-Verdi, rather than some secure subset of an existing popular language like Pascal or Ada. In order to prove assertions about any existing

software with m-EVES, translators will first need to be written to translate the software into m-Verdi, as is done with systems in which the input language cannot be executed, like MALPAS. This of course raises the issue of whether the translators need to be verified.

3 m-NEVER

m-NEVER is the interactive theorem proving component of the m-EVES system. m-NEVER is a powerful theorem prover. There are very few commands available to the user, and, although it only takes a short time for a new user to learn each of these commands, it can require a high level of experience to use them to solve nontrivial problems.

Typically the user wishes to prove the correctness of a program. Declarations from the program are entered one at a time. Each declaration's proof obligation becomes the current formula. The user may prove the current formula, or may move on to the next declaration. Some simple declarations, like those of unstructured variables, have no proof obligation.

In order to successfully discharge some proof obligations it may be necessary to make further declarations for m-NEVER's use, introducing facts, rewrite rules and forward rules. These declarations, described below, will have proof obligations of their own.

3.1 Declarations

Just as the distinction between m-Verdi and m-NEVER is blurred to the user, so is the distinction between the program and its proof. For example, if a user wished to use a variable in a rule definition then that variable would first need to be declared in the same way as if for use in a procedure declaration. All the forms of definition described in the section on m-Verdi above are still relevant to m-NEVER, in addition to some new forms of definition. Strictly speaking, the syntax of these definitions is defined by m-Verdi, but as they are relevant only for use with m-NEVER they will be described here.

3.1.1 Facts

Facts are used to give names to axioms that a user may wish to use while proving a theorem. The declaration of a fact gives rise to a proof obligation that the axiom is in fact true. Users need not immediately discharge proof obligations. Indeed, in some cases, this may never be possible.

A fact may be parameterised, in which case the fact is implicitly universally quantified over those variables by which it is parameterised. In general, parameterised facts are used in preference to explicitly quantifying the variables, because the commands available in m-NEVER allow better use to be made of parameterised facts.

Below is an typical fact definition:

```
axiom gcdsym(i, j) =
begin
  implies(not(and(i = 0,
                  j = 0),
          gcd(i, j) = gcd(j, i))
end gcdsym
```

3.1.2 Rewrite Rules

Rewrite rules are parameterised axioms used by m-NEVER's rewriting tactic. The body of a rewrite rule must take the following form:

`implies(`*`expression, expression = expression`*`)`

The rewriting tactic searches the current formula for expressions which match the left hand side of the equality in a rewrite rule. Matching is performed by instantiating the parameters of the rule. If an expression is matched, and m-NEVER can simplify the condition of the implication in the rule to `true`, then that expression will be replaced by the right hand side of the equality.

When the body of a rewrite rules is of the form:

`implies(`*`true, expression = expression`*`)`

it may be abbreviated to:

`expression = expression,`

Such rules are known as *unconditional* rewrite rules, while other rewrite rules are known as *conditional* rewrite rules. Similarly, if the equality expression in a rewrite rules is of the form:

`expression = true,`

it may be abbreviated to:

`expression,`

A typical rewrite rule looks something like the one below:

```
axiom gcdsub(i, j) =
rule
begin
  implies(not(and(i = 0,
                  j = 0)),
          gcd(minus(i, j), i) = gcd(i, j))
end gcdsub
```

The interaction below is an example of the use of this rewrite rule:

```
        .
        .
        if x = 0 then
                .
                .
        else
                .
                .
                gcd(minus(x, y), x)
                .
                .
        end if
        .
        .
! rewrite;

Which simplifies
when rewriting with ... GCDSUB ... to ...
        .
        .
        if x = 0 then
                .
                .
        else
                .
                .
                gcd(x, y)
                .
                .
        end if
        .
        .
```

Note that the nature of the rewrite tactic means that, even if all other rules are disabled, the formula resulting from rewriting is unlikely to bear as close a resemblance to the original as suggested above.

3.1.3 Forward Rules

Forward rules are parameterised axioms used by m-NEVER's rewriting tactic. The body of a forward rule must take the following form:

`implies (`*`expression, expression`*`)`

The definition of a forward rule also contains a list of trigger expressions. The rewriting tactic attempts to match expressions in the current formula with trigger expressions from forward rules. Matching is performed by instantiating the parameters of the rule. Should a trigger expression be matched, the same instantiation will be applied to the body of its rule. If m-NEVER can simplify the condition of the instantiated rule to `true`, then the consequent will be added to the list of hypotheses with which m-NEVER is rewriting.

When the body of a forward rules is of the form:

`implies (true, expression)`

it may be abbreviated to:

expression

Such rules are known as *unconditional* forward rules, while other forward rules are known as *conditional* forward rules.

Forward rules are like facts, in that they are used to add more assumptions to the list of hypotheses that m-NEVER is using to rewrite the current formula. Facts typically add an assumption which is a general predicate on universally quantified variables. Forward rules, on the other hand, typically add an assumption in which the variables are instantiated with a particular valuation relevant to the formula being simplified. Use of an assumption from a forward rule is more efficient than using one from a fact, since it avoids instantiating the assumption on every use. The user also gains more control over how the assumption is used by restricting it to a particular valuation.

Once included, a fact is used in every rewriting. If the fact is irrelevant in a particular rewriting, then attempts to make use of it will lead to unnecessary work on the part of m-NEVER. Forward rules are used to add relevant assumptions dynamically when rewriting a formula (A forward rule is deemed relevant if one of its trigger expressions appears in the formula.).

A forward rule from the problem in appendix A appears below as an example:

```
axiom containsmember(a, p, e) =
frule
triggers (atype'extract(a, p) = e)
begin
  all i, j : implies(and(int'ge(i, a'lowbound),
                         int'ge(p, i),
                         int'ge(j, p),
                         int'ge(a'highbound, j),
                         atype'extract(a, p) = e),
                     contains(a, i, j, e))
end containsmember
```

The interaction below is an example of this forward rule:

```
  .
  .
  int'ge(i, a'lowbound)
  .
  .
  int'ge(p, i)
  .
  .
  int'ge(j, p)
  .
  .
  int'ge(a'highbound, j)
  .
  .
  atype'extract(a, p) = e
  .
  .
  contains(a, i, j, p)
  .
  .

! rewrite;

Which simplifies
with the assumptions ... CONTAINSMEM ... to ...
  .
  .
  int'ge(i, a'lowbound)
  .
  .
  int'ge(p, i)
  .
  .
  int'ge(j, p)
  .
  .
  int'ge(a'highbound, j)
  .
  .
  atype'extract(a, p) = e
  .
  .
```

Once again, the nature of the rewrite tactic means that the formulae resulting from the use of forward rules can differ significantly from the original.

3.2 Commands

There are several commands to control the basic state of m-NEVER. In particular, commands are

available to commence work on proofs, to save and restore the state, and to undo proof steps. Of more interest are those commands used to manipulate the formula for which a proof is being attempted. This formula is known as the *current formula*. Commands for manipulating the current formula are described below.

3.2.1 Simplification

`simplify`

This command invokes m-NEVER's simplification heuristics. Simplification finds propositional tautologies, substitutes expressions for other equal expressions and instantiates quantified variables.

`simplify without case analysis`

This command performs a subset of what the `simplify` command does. Namely, if a function call contains a conditional expression the conditional is not moved out of the function call.

`simplify without instantiation`

This command simplifies the formula without making any instantiations of quantified variables. Of the simplification commands this one probably represents the optimal compromise between speed and results.

`trivial simplify`

This command checks for simple tautologies. In practice most tautologies are too hard for `trivial simplify` to identify.

3.2.2 Rewriting

`rewrite`

The formula is simplified using the rewrite and forward rules, simplification and by expanding functions to their definitions.

When using the `rewrite` command, m-NEVER lists the names of those rewrite rules used in the rewriting and those forward rules whose patterns were matched during rewriting. It would be more useful if a distinction was made between those forward rules which matched and contributed information which led to a simplification, and those forward rules which only matched.

`trivial rewrite`

The formula is rewritten using only the unconditional rewrite rules.

3.2.3 Substitution

`invoke`

Invocation is used to substitute a function call with its definition. The `invoke` *`name`* command will replace every instance of the function *`name`* in the formula with its definition. In order to expand particular instances of a function application the `invoke` *`name(arguments)`* variant of the command is used, which only expands those instances of the function with the arguments specified. If there is more than one instance of a function application with the same arguments there is no way to expand a particular instance.

In the case of partial functions (functions with preconditions) the function is replaced by a conditional expression equal to the function's definition only if the precondition is true. Consider the generic partial function below:

```
function F(a) : T =
pre P[a]
begin
  E[a]
end F
```

Now if the current formula contained the expression `F(x)` then invoking `F` would replace `F(x)` by:

```
if P[x] then
  E[x]
else
  F(x)
end if
```

This is intended to mean that if the precondition holds, then the function is equal to its defining expression; otherwise nothing more is known about the value of the function than before.

equality substitute

Equality substitution is used to replace expressions in the formula with other expressions to which they are equal. If the formula contains an implication and the antecedent of the implication is (or has as one of its conjuncts) the expression $e_1 = e_2$ or $e_2 = e_1$ then the command `equality substitute` e_1 will replace all instances of e_1 in the consequent with e_2. Similarly if the condition of an conditional statement is (or has as one of its conjuncts) the expression $e_1 = e_2$ or $e_2 = e_1$ then the command `equality substitute` e_1 will replace all instances of e_1 in the then part of the expression with e_2. It is not possible to use equality substitution to repace particular instances of an expression.

If an expression `e` appears in multiple equality expressions which are in the antecedent of an implication or the condition of a conditional then it is not possible to use `equality substitute` to substitute new values for `e` in only one of the expressions in which it appears.

The equality substitute command may be given a list of substitutions to make, as in: `equality substitute` e_1, e_2... If the list of substitutions is the empty list then m-NEVER heuristically chooses which substitutions to make.

label

The command `label` *expression* replaces every occurence of *expression* in the current formula with a label of m-NEVER's choosing and the predicate *identifier* = *expression*, where *identifier* is the chosen label, is made an additional hypothesis of the current formula. Labelling is used to reduce the complexity of an expression which contains a complex and repeated subexpression.

3.2.4 Reduction

reduce

This command attempts to reduce the current formula through simplification, rewriting and invocation.

reduce without instantiation

This command attempts to reduce the current formula though simplification without instantiation, rewriting and invocation.

3.2.5 Definitions

use

The function of the `use` command is to incorporate a particular fact, rewrite rule or forward rule into the current formula as a hypothesis. The `use` command conforms to the following template:

`use` *name* p_1 = e_1, p_2 = e_2, ...

where *name* is the name of a fact, rewrite rule or forward rule, the *p*s are parameters of the named axiom and the *e*s are expressions. The named axiom is added to the current formula as a hypothesis with those parameters nominated in the command instantiated with the expression specified. Any unused parameters are universally quantified.

prove by facts

This command adds all the available facts to the formula, as if with the use command, and then attempts to reduce the resulting formula.

enable, disable

Rewriting, reduction and proving from facts automatically make use of definitions in the prover's database. Rewriting uses rewrite rules and forward rules. Reduction uses these and function definitions as well. The user has the option of enabling or disabling these various definitions with the `enable` and `disable` commands which conform to the following template:

`(enable | disable)` *name*

where *name* is the name of the definition in question.

3.2.6 Case Analysis

split

The `split` command is used to partition the current formula about a given predicate. If the current formula is *F* then partitioning about the predicate *P* results in the new current formula:

if *P* **then**
  F
else
  F
end if

which allows consideration of *F* in the case when *P* is true, separately from the case where *P* is false.

prove by cases

This command attempts to reduce each conjunct or disjunct of the current formula independently of the others. The result is the recombination of each of the subproof results with the appropriate conjuncts and disjuncts.

3.2.7 Induction

induct

The `induct` command is used to employ a Boyer-Moore style induction scheme on the current formula. The `induct` command heuristically chooses a term on which to induct by default. However, the user may specify a specific term by using the syntax:`induct on` *term*.

prove by induction

This command is similar to the `induct` command except that, having chosen the induction scheme as with `induct`, `prove by induction` goes on to reduce the formula.

3.2.8 Normal Forms

conjunctive

This command recasts the current formula in conjunctive normal form.

disjunctive

This command recasts the current formula in disjunctive normal form.

rearrange

Rearranging the current formula by moves simpler expressions to the front of conjunctions and disjunctions. Rearranging makes a formula more amenable to reduction.

prove by rearrange

This command attempts to prove the current formula by rearranging it. This command consists of reduction without instantiation, rearranging and simplification without case analysis.

3.2.9 Quantifiers

open

Opening uses specialisation to remove enclosing universal quantification from the current formula.

close

Closing the current formula uses generalisation to universally quantify free variables in the current formula, thus forming its *universal closure.*

prenex

`prenex` recasts the formula in prenex normal form, i.e. with quantifiers having the maximum possible scope. This is frequently a precursor to using the `open` command.

instantiate

This command, when accompanied by a list of instantiations, is used to instantiate quantified variables in the current formula.

3.2.10 Counter Examples

counter

This command attempts to find counter examples to the current formula. The universal closure of the current formula is formed as with the `close` command and the resulting formula is reduced. `counter` is considered successful if it returns `false` for any instantiations of the resulting formula.

3.2.11 State Manipulation

reset

This command resets the theorem prover's state to the initial form it had immediately after start-up.

freeze

The command `freeze "`*name*`"` stores the theorem prover's current state in the file *name*.

thaw

The `thaw "`*name*`"` command restores the theorem prover's current state to the state stored in the file *name*.

try

A user may make another definition after commencing the proof of the previous definition's correctness, leaving behind a partial (possibly empty) proof. The command `try` *name* is used to return the theorem prover to the partial proof of the definition *name*. The command `try` *P*, where *P* is some predicate, is used to commence *ad hoc* proofs.

3.3.1 Strengths of m-NEVER

m-NEVER is a powerful theorem prover capable of proving nontrivial theorems with little or no assistance from the user. Indeed, m-NEVER is possibly the most powerful theorem prover of its type.

Only a few simple commands are required to operate m-NEVER. A new m-NEVER user will be able to prove nontrivial theorems within a relatively short space of time. In contrast, systems like HOL require much learning and skill on the part of the user before proofs can be successfully completed. For this reason m-NEVER, more so than many other theorem provers, may be suited to applications in industry where a high degree of formal training cannot be expected of users.

m-NEVER is an interactive theorem prover. This gives the user the ability to guide proofs as they progress. This ability is a significant advantage over batch oriented systems, like the Boyer-Moore theorem prover and MALPAS's algebraic simplifier, in which users must attempt to supply all the information the prover may require before the proof is commenced.

The inclusion of forward rules, a facility not available in MALPAS, can greatly enhance the automatic capabilities of m-NEVER. It can, however, require a significant level of experience to determine whether an axiom is more usefully expressed as a rewrite or forward rule.

3.3.2 Weaknesses of m-NEVER

m-NEVER's simple proof management scheme makes it difficult to use lemmas in proofs. It is common practice among people hand proving formulae to assume various lemmas as they are required. These lemmas are then discharged after the conclusion of the original proof. For example, a person attempting to prove a formula, f, may, during the proof, assume a lemma, l; returning to prove l only after finishing the proof of f. The existence of the `try` command suggests that this style of reasoning would be supported by m-NEVER. Below is a sample interaction showing what happens if this is attempted:

User declares formula to prove:

```
! axiom f(v1, v2,...) =
  begin
    P1 (v1, v2,...)
  end f;
F
Beginning proof of F ...
P1 (V1, V2,...)
```

User simplifies the proof obligation:

```
        :

P'1 (V1, V2,...)
```

User declares lemma to help complete proof:

```
! axiom l(v1, v2,...) =
    begin
      P2(v1, v2,...)
    end l;
L
Beginning proof of L ...
P2 (V1, V2,...)
```

User returns to proof of formula:

```
! try f;
Continuing proof of F :
P'1 (V1, V2,...)
```

User attempts to use lemma:

```
! use l;
```

m-NEVER has forgotten the lemma:

```
!
```

This problem occurs because when the `try` command is used to return to some partial proof the system is returned to the state it was in when that proof was last current. The motivation for this may be as a naive method for avoiding circular reasoning. This feature has a strong impact on the way in which m-NEVER is used. Before commencing a proof a user may declare several axioms on the chance that these axioms will be useful during the proof. However, it is not always possible to tell which axioms will be needed before a proof has begun, or whether they will be more useful if expressed as facts, rewrite rules or forward rules. If while constructing a proof the user realises that a lemma is needed then the current proof must be undone, that required lemma declared and the proof recommenced. In cases where it is necessary to undo a large amount (several hours) of work to declare a lemma a user may wish to prove the lemma before using it.

m-NEVER only supports reasoning in one particular logic. This does not present a problem if m-NEVER is only used as a tool for reasoning about m-Verdi programs. However, if users wish to employ m-NEVER to reason about questions from other problem domains then they may find that the supported logic does not afford them the most natural way of expressing themselves. This limitation has become a weakness of the HOL system which also supports a single logic. HOL's logic is natural for reasoning about hardware problems, its intended problem domain. However, as the power of this theorem prover has been recognised, people have tried to apply it to problems from other domains, like software verification, and have found that problems from these other domains are less naturally expressed in HOL's logic. In contrast, Isabelle [14] and μ**ral** [9] have been designed as generic theorem provers which can use multiple logics.

Only a fixed set of tactics, hard-coded into the theorem prover, are supported by m-NEVER. Other systems, such as HOL and Isabelle, allow users to write their own tactics. The ability for users to create their own tactics is important as it allows users to assemble a set of commands well suited to solving problems in each theory they develop. While m-NEVER does allow the user to declare rewrite and forward rules for use with its built-in tactics, this facility is much less powerful than allowing users to write their own tactics.

Only a small set of commands, allowing a coarse level of interaction, are supported by m-NEVER. Interactivity should not be regarded as a boolean valued property of theorem provers. Theorem provers can be placed on a spectrum of interactivity. At one end of the spectrum are theorem provers like Demo2 [17], which allow the user to manipulate formulae in the most intricate ways possible. At the other end of the spectrum lie systems like MALPAS and the Boyer-Moore theorem prover, where the only option available to the user is whether to invoke the theorem prover or not! Systems like m-NEVER and GVE [18] lie somewhere between the two extremes. m-NEVER offers only a few coarse grain commands to the user. The optimal situation is to have a theorem prover with commands that allow very fine-grained manipulation of predicates, and the ability to combine these commands with tactics to create more powerful, but less fine-grained, commands. To use an analogy: if you wish to make shapes out of stone, it is better to have a pile of gravel and some glue than it is to have a boulder and no chisel. m-NEVER represents a middle ground where you have several preformed rocks, which might not be of the right shape, but are closer than the boulder, and did not require assembling.

The coarse-grained nature of m-NEVER's commands for manipulating predicates can be frustrating to a user. It seems to be assumed that most of the intelligence required for theorem proving is embodied in the heuristics used by m-NEVER and this assumption has led to a system in which it is difficult for users to express their own intelligence and intuition. Users of m-NEVER will often see simplifications that they would like to make, but will find it difficult to state how to make the desired simplification in terms of commands like `simplify`, `reduce`, etc. To use another analogy: if you are asked to carve a statue from the stone in a cliff face you will find a range of tools useful. Dynamite may let you get a piece of stone of about the right size. A jackhammer might get the stone to the right shape. You may wish to add the finishing touches with a chisel. Sometimes, using m-NEVER is like trying to add the finishing touches to a statue with a jackhammer. The coarse nature of the commands also makes m-NEVER a slow system. Even if the user only wishes to make a small transformation to the current predicate, m-NEVER invariably attempts a more significant simplification, even if none can be made, taking several minutes to complete the desired action.

It is difficult for users of m-NEVER to focus their attention on subproblems within the current formula. When faced with a large problem a common method for dealing with the complexity is to focus attention on some subproblem. Although this is a natural human problem solving strategy it is not supported by m-NEVER. There is no way to set the current formula to some subexpression of the formula being worked on. All m-NEVER commands affect the entire current formula. This means that any command issued with the intention of simplifying some subexpression may significantly complicate other parts of the current formula. Indeed, after issuing some commands it may be difficult to tell which portions of the resulting predicate correspond to the subexpression the command was intended to simplify. The theorem prover Demo2, a system based around the concept of opening windows on subexpressions, demonstrates the power of a system in which the user and the theorem prover can restrict their attention to a particular subexpression.

Although it is not possible to focus m-NEVER's attention on a subexpression of the current formula, this is such a fundamental theorem proving technique that it is necessary to simulate this approach in order to analyse complex predicates. Typically m-NEVER users identify subexpressions of the current formula they wish to simplify, then recommence the proof after declaring rules designed to simplify the chosen subexpressions. This process can be very slow. Also, this technique is rather different to a more traditional hand proving approach where people may work through each subexpression, usually only defining separately those lemmas which are considered to be of general or repeated use. A complex proof in m-NEVER will require several definitions which will only ever be of use for that particular proof. The necessity to define many problem

specific rules is also true of MALPAS. The problem is exacerbated in MALPAS where the user is not encouraged to prove that the defined rules are correct.

The transformations made to the current formula by m-NEVER's commands often leave it in an unrecognisable form. The form of a predicate has a dramatic effect on a human's ability to think and reason informally about it. For example, the sentence "If it does not rain today I will go for a walk and wash the car." can be expressed by the predicate: $\neg R \Rightarrow W \wedge C$, where R denotes "it rains today", W denotes "I go for a walk", and C denotes "I wash the car". The sentence can also be expressed by the equivalent predicate: $(R \vee W) \wedge (R \vee C)$. The first predicate, though only slightly simpler than the second, is significantly easier for a human to reason about, as it more closely resembles the way a human would state the problem. Users typically spend some time phrasing specifications in ways that are intuitively easy for them to reason about. Many of the m-NEVER commands transform predicates in such a way that they become unrecognisable and are no longer natural to reason about. It may take a user several minutes to understand the new predicate so as to ascertain whether it represents a significant simplification of the previous one. This problem is related to the coarse nature of m-NEVER commands and to the inability to restrict their effect to subexpressions of the current formula.

m-NEVER's heuristics are particularly good in some areas, such as verifying iterative solutions against recursive specifications, but can be weak in others. This can lead to a tendency to specify problems in a manner that suits the theorem prover rather than the specifier. The purpose of specification is to capture, in a formal manner, those concepts in a human's mind that form their requirements for a program. It is the purpose of verification to guarantee the correctness of a program with respect to a specification. However, there can be no guarantee that a program's specification captures the intent of its human specifier. One way to reduce the chance of errors in a specification is to specify in an abstract manner. Specifying abstractly decreases the semantic gap between the abstract concepts in the specifier's mind and the specification, thereby increasing the probability that the specification will be correct. It also increases the semantic gap between the specification and the code, thereby increasing the difficulty of verifying the code. Nonetheless, specifications should remain as abstract as possible so as to increase the probability of their correctness. There is no point in easing the difficulty of verifying code against a specification if this is achieved by bringing the correctness of the specification into doubt. For example, suppose a user wishes to develop a program to find the greatest common divisor of two natural numbers. One specification could be that a common divisor of two numbers is a number that divides both numbers exactly, the greatest common divisor being the greatest of all such numbers. This specification is shown formally below.

$$div\ z \triangleq \{x | z\ mod\ x = 0\}$$
$$\mathrm{GCD}\ x\ y \triangleq max\ (div\ x \cap\ div\ y)$$

Another specification could be that the greatest common divisor of a number and itself is that number, and the greatest common divisor of two distinct numbers is the greatest common divisor of the smaller one and the difference between them. This specification is shown formally below:

$$\mathrm{GCD}(x,y) \triangleq \begin{cases} x & ,x = y \\ \mathrm{GCD}(\min(x,y), \max(x,y) - \min(x,y)) & ,x \neq y \end{cases}$$

It is by no means obvious that the second specification (actually Euclid's algorithm for calculating greatest common divisors) specifies the greatest common divisor of two numbers. The task of verifying an iterative implementation of Euclid's algorithm against the second specification is, of course, much easier than verifying it against the first, more abstract, specification. However,

verifying an implementation against the second specification is not a worthwhile task unless it is proven that the two specifications are equivalent. A recursive specification does not always represent the most abstract way to specify a property, as was the case with the example above. m-NEVER users would be well advised to avoid the tendency to choose specifications against which they will be able to verify their programs with ease over specifications which transparently encapsulate the properties they desire of the program under development. This tendency is displayed in the m-EVES documentation. The m-NEVER distribution includes an example verification of an iterative version of Euclid's greatest common divisor algorithm against a recursive specification of Euclid's algorithm. Further, in the paper *The TR Program Example* [10], included as part of the m-EVES documentation, the following strategy for producing *verified* programs is recommended:

> A good verification strategy is to implement something iteratively, and specify it recursively. Then, prove that the specification meets the implementation. This is the basic technique used throughout the `tr` program. I once tried to specify the character map and initialization procedures by using quantifications to describe the results of the procedures. This was a bad idea on two counts:
>
> - The specification was descriptive, not constructive.
> - It is more difficult to prove VC's that contain quantifications. The prover has better heuristics for recursive definitions.

This philosophy — where a program is written, and a specification then chosen to fit — is one in which the cart has been placed firmly before the horse. If the semantic gap between a specification and an implementation is too great to prove their correspondence then a good approach is to choose some intermediate specification, prove the equivalence of the two specifications, and verify the program against the intermediate specification. This approach can be very demanding, highlighting the difficulty of proving nontrivial theorems with m-NEVER. Trivialising programs' specifications in order to simplify their proof obligations is not the correct way to make this difficulty go away.

m-NEVER, as part of m-EVES, is not freely distributed, and as a tool with a restricted user base m-NEVER is destined to have a restricted development path and limited support. A system with a strong user base and for which the source code is freely available will evolve and improve at the hands of its users. For example, HOL is now being rewritten by its users at the University of Calgary. In contrast, the evolution of m-NEVER will likely only take place in the direction and at the rate chosen by Odyssey Research Associates. The more users any tool has, the more experience will be gained with it, and the more supporting tools will be developed for it. For example, HOL is freely available and widely used. As a result, a number of libraries of theories and tactics written by users have become part of the HOL distribution. Also, a tool to provide a limited SunView™ interface has also been developed by a HOL user. Users of popular systems frequently band together to form self-help groups. The `haskell` mailing list, news groups like `comp.text.tex` and user groups like HOL's are all examples of popular forums where experiences can be shared and valuable lessons learned. m-NEVER's small user base means that this level of support is unlikely to develop.

4 DOCUMENTATION

The documentation for the m-EVES system is the book *m-EVES Collected Papers* [11]. This book is a collection of the following papers: *m-EVES: A Tool for Verifying Software* [4], *An Application of the m-EVES Verification System* [3], *Formal Verification in m-EVES* [13], *A Description of m-Verdi* [2], *An m-Verdi User's Guide* [5], *m-EVES User's Manual* [12] and *TR Program Example* [10]. Each of these papers is of a very high standard and more than adequate in its original role as either a technical report or a conference paper. However, it is not the case that collectively these papers represent adequate documentation for m-EVES.

The papers *A Description of m-Verdi* [2] and *An m-Verdi User's Guide* [5] form, in combination, an adequate reference manual for m-Verdi. A good reference manual is essential for both the novice and experienced user of any complex system. These two papers have a similar structure and cover the same breadth of material. In some cases the same information is contained in both papers, more or less verbatim. In other cases, each paper contains information which is not contained in the corresponding section of the other paper. To understand a particular feature of the m-Verdi language it is usually necessary to consult both papers. The m-EVES documentation would benefit greatly if these two papers where replaced by one comprehensive reference on m-Verdi.

A Description of m-Verdi [2] and *An m-Verdi User's Guide* [5] describe a version of m-Verdi that is somewhat different to the version implemented in m-EVES version 4. Both papers describe language features, like packages and environments, which are not supported by m-EVES. In other cases, the language features described in the papers are supported by m-EVES, but with a different syntax. For example, below is an axiom definition example from *A Description of m-Verdi* [2]:

```
axiom
    pragma(name = "sequence_6")
    all s,m:  and(length(new_sequence_of_message) = 0,
                  length(apr(s,m)) = plus(length(s),1),
                  Int'ge(length(s), 0))
```

This axiom should instead be written:

```
axiom sequence_6(s, m) =
begin
  and(length(new_sequence_of_message) = 0,
      length(apr(s, m)) = plus(length(s), 1),
      int'ge(length(s), 0))
end sequence_6
```

The only way for users to learn the correct syntax for these language features is to look at examples in the more recent papers. This level of inaccuracy diminishes the worth of *A Description of m-Verdi* [2] and *An m-Verdi User's Guide* [5] as works of reference.

m-EVES User's Manual [12] provides the reference material for m-NEVER. A later version of this paper forms the on-line documentation of m-EVES. The differences between the on-line and textual versions of the paper are subtle. Two important differences are that the on-line version does not state that procedure stubs are not supported, and the on-line version describes a library mechanism that is not discussed in the textual version. It would be of benefit to all users if the latest version of this paper were part of *m-EVES Collected Papers* [11].

In general the descriptions found in *m-EVES User's Manual* [12] are accurate, with the exception of continual references to a nonexistent menu system. For example the following is an extract from the manual decribing the `enable` command:

```
ENABLE
enable [RULE-OR-NAME] ;

Enables the given RULE-OR-NAME. Once enabled, the automatic
prover can use it. If RULE-OR-NAME is not supplied then a menu of
currently disabled events is produced.
```

This menu feature may be supported on the version of m-EVES for Symbolics™ Lisp machines,

but it is definitely not present in the version for Sun™ work stations.

Sorely missed from the reference material on m-NEVER is information, not on what each of the commands does, but rather in which circumstances each command is useful. For example, while the manual explains the syntax of rewrite and forward rules, and how m-NEVER uses them, no attempt is made to explain the class of problems that these rules are used to solve. This sort of practical information on how to use the commands to generate successful proofs would be of great benefit to new users. Some knowledge on how to best use the commands can be gained from *m-EVES User's Manual* [12] and from the proof examples in the other papers, but mostly it must be acquired though trial and error. Were the collective wisdom of m-NEVER's developers on how to use the commands to be documented, the resulting information would greatly speed the process of becoming a competent m-NEVER user.

Also missing from m-EVES' documentation is a tutorial-style introduction to the system. While the papers give several general overviews of the system and a few worked examples, they do not provide a gentle and thorough introduction to using the system. Such a tutorial, while not being the original intent of the papers, is a necessary part of good documentation. A tutorial guide to m-EVES should introduce each of the features of m-Verdi and m-NEVER and include a series of examples to ensure the user understands the feature introduced.

In summary, while each of the papers in *m-EVES Collected Papers* [11] is fine, they do not, as a collection, represent adequate documentation for the m-EVES system. A comprehensive and detailed tutorial guide and reference manual are needed to replace the existing documentation. This lack of adequate documentation greatly increases the time it takes to become competent at using the system. Although m-EVES' ease of use makes it ideally suited to compete with systems like MALPAS in the commercial verification market, it will first need to be properly documented, as commercial software companies can ill afford to expend the time required to learn their tools by trial and error.

5 CONCLUSIONS

In this report, m-EVES has been examined as an environment for the development of formally verified software. The discussion has centred on its two main components: the language, m-Verdi, and the theorem prover, m-NEVER, and comments have been made on their strengths and weaknesses. The documentation has also been discussed.

It has been seen that m-Verdi has a simple semantics, while having a high degree of expressiveness as a programming language. It includes some quite useful constructs to aid in theorem proving, and has facilities for modelling a program's target hardware. However, the language is verbose and cumbersome in appearance, making it somewhat difficult for the user to analyse large predicates. In the author's opinion, the lack of polymorphism, and the inability to define recursive procedures, are severe and unnecessary restrictions on the language.

The theorem prover, m-NEVER, has also been examined, revealing the following strengths:

- m-NEVER is a powerful theorem proving tool, capable of automatically proving non-trivial theorems;
- its small number of theorem proving commands means that a new user is able to prove theorems after just a short introduction to m-EVES;
- the ability to use forward rules is very useful; and
- m-NEVER is interactive, allowing the user to guide the proof as it progresses.

However, some weaknesses which limit m-NEVER's usefulness were also revealed:

- its simple proof management scheme makes it hard to use lemmas in proofs;
- reasoning in only one logic is supported;
- users cannot define their own proof commands (tactics);
- there is just a small set of built-in proof commands, which are quite coarse-grained and can be time-consuming;
- the prover cannot focus on subexpressions of the predicate one is trying to prove; and
- m-NEVER's heuristics are good in some areas but weak in others, and can lead to a tendency to specify problems in a manner that suits the theorem prover, rather than the specifier.

It is important, in evaluating program verification environments, to compare how non-trivial problems can be solved within different systems. In further work, it is planned to carry out such a comparison between m-EVES, Gypsy, MALPAS and HOL. Crucial points to address in such a study would be:

- the user interface of the tool;
- the expressiveness and flexibility of the input language, the theorem proving commands etc;
- the time it takes a new user to become comfortable with the system, and to be able to prove significant results;
- the time and effort required by the experienced user to formulate a problem, and derive the machine proof;
- the speed of the theorem prover itself; and
- if the tool has deficiencies, how easy it is to remedy them.

6 ACKNOWLEDGMENTS

The author would like to thank the following members of the Trusted Computer Systems Group, Information Technology Division, ERL: Dr Brian Billard, Dr Tony Cant, Katherine Eastaughffe and Maris Ozols for discussions and their many helpful comments on earlier drafts of this report. Dr Cant helped greatly with the completion of the final version.

The author is especially grateful for the comments provided by the referees: Dr Stephen Crawley (Information Technology Division) and Professor John Staples (Department of Computer Science, University of Queensland).

m-EVES has been made available by Odyssey Research Associates (Canada) under the auspices of TTCP Technical Panel 1 of Subgroup X, for which appreciation is expressed.

REFERENCES

[1] Robert Boyer and J. Strother Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press — Harcourt Brace Jovanovich, Boston, United States, 1988.

[2] Dan Craigen. A description of m-Verdi. In *m-EVES Collected Papers*. I.P. Sharp Associates, Ottawa, Canada, 1987.

[3] Dan Craigen. An application of the m-EVES verification system. In *m-EVES Collected Papers*. I.P. Sharp Associates, Ottawa, Canada, 1988.

[4] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Andy Neilson, Bill Pase, and Mark Saaltink. m-EVES: A tool for verifying software. In *m-EVES Collected Papers*. I.P. Sharp Associates, Ottawa, Canada, 1987.

[5] Dan Craigen and Mark Saaltink. An m-Verdi user's guide. In *m-EVES Collected Papers*. I.P. Sharp Associates, Ottawa, Canada, 1987.

[6] Donald I. Good, Robert L Akers, and Lawrence M Smith. Report on the Gypsy 2.05. Technical report, Computational Logic Inc, Austin, United States, 1989.

[7] Michael J.C. Gordon, editor. *The HOL System*. SRI International — Cambridge Research Centre, Cambridge, United Kindgom, 1989.

[8] Cliff Jones. *Systematic Software Development Using VDM*. Prentice-Hall International Series in Computer Science. Prentice/Hall, Englewood Cliffs, United States, 1986.

[9] Peter A. Lindsay. The **μral** proof assistant. In *Australian Computer Science Conference 13*, 1990.

[10] Irwin Meisels. TR program example. In *m-EVES Collected Papers*. Odyssey Research Associates, Ottawa, Canada, 1989.

[11] Odyssey Research Associates, Ottawa, Canada. *m-EVES Collected Papers*, 1989.

[12] Bill Pase and Mark Saaltink. m-EVES user's manual. In *m-EVES Collected Papers*. I.P. Sharp Associates, Ottawa, Canada, 1987.

[13] Bill Pase and Mark Saaltink. Formal verification in m-EVES. In *m-EVES Collected Papers*. I.P. Sharp Associates, Ottawa, Canada, 1988.

[14] Lawrence C. Paulson and Tobias Nipkow. *Isabelle Tutorial and User's Manual*. Computer Laboratory, University of Cambridge, Cambridge, United Kingdom, 1990.

[15] Rex Thompson & Partners, Surrey, United Kingdom. *MALPAS Users Guide*, 1990.

[16] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, New York, United States, 1989.

[17] Tong Gao Tang, Peter J. Robinson and John Staples. The demonstration proof editor Demo2. Technical report, Key Centre for Software Technology, Department of Computer Science, University of Queensland, St Lucia QLD 4067, Australia, 1990.

[18] William D. Young. GVE user's manual: Examples of proof commands. Technical Report 8, Computational Logic, Austin, United States, 1987.

APPENDIX A BINARY SEARCH PROBLEM

This appendix describes the development of an m-Verdi algorithm for binary searching sorted arrays. To binary search a sorted array, for a particular target, one looks at the midpoint of the array. There are three possibilities: either the midpoint is the target and the search is complete; or the midpoint will be greater than the target, in which case the search can be restricted to the lower half of the array; or the midpoint will be less than the target, in which case the search can be restricted to the upper half of the array. If the target has not been found, then the portion of the array still under consideration is cut in half again, and so on. This process continues until the target is found or the array is exhausted. This algorithm is an important one as it can search an array in $O(\log n)$ time (where n is the size of the array).

Several integer variables are introduced for use throughout the program and its supporting axioms.

```
var i, j, k, l, m, t : int;
```

A function for calculating the mid-point of two indices is introduced. This is done through a function stub and an axiom asserting the defining properties of the function. It would have been more natural to define the function as calculating the average of the two indices; however, this definition proved difficult to work with.

$$\frac{j \geqslant i}{mid(i,j) = \left\lfloor \frac{i+j}{2} \right\rfloor}$$

```
function mid(i, j) : int;

axiom middef(i, j) =
frule
triggers (mid(i, j))
begin
   implies(int'ge(j, i),
          some l : and(plus(i, l) = mid(i, j),
                       or(plus(mid(i, j), l) = j,
                          plus(plus(mid(i, j), l), l) = j)))
end middef;
```

The proof obligation of this axiom is not discharged as it is `mid`'s definition and not some provable property of the `mid` function.

Two rewrite rules are used to express the fact that the mid-point of two indices lies between them. These rules and their proofs are given below:

$$\frac{j \geqslant i}{mid(i,j) \geqslant i}$$

```
axiom midincl(i, j) =
rule
begin
   implies(int'ge(j, i), int'ge(mid(i, j), i))
end midincl;

use middef i = i, j = j;
prenex;
simplify;
```

$$\frac{j \geqslant i}{j \geqslant mid(i,j)}$$

```
axiom midinc2(i, j) =
rule
begin
   implies(int'ge(j, i), int'ge(j, mid(i, j)))
end midinc2;

use middef i = i, j = j;
prenex;
simplify;
```

If the mid-point of two indices is the same as the (supposedly) larger of the two, then the two indices are the same.

$$\frac{j \geqslant i \, , j = mid(i,j)}{i = j}$$

```
axiom midprop(i, j) =
frule
triggers (j = mid(i, j))
begin
   implies(and(int'ge(j, i),
               j = mid(i, j)),
           i = j)
end midprop;
```

The proof of this property follows directly from `mid`'s definition.

```
use middef i = i, j = j;
prenex;
split i = j;
simplify;
```

Two integer constants, `top` and `bot`(tom), are introduced. Such declarations have no proof obligations. It is asserted that `top` is at least as big as `bot`. This assertion is a statement of fact and no attempt is made to meet its proof obligation.

bot : **Z**

```
const bot : int;
```

top : **Z**

```
const top : int;
```

bot $\leqslant$ top

```
axiom botletop() =
frule
triggers (bot, top)
begin
   int'le(bot, top)
end botletop;
```

An array type vec is declared as integer arrays from bot to top.

```
type vec = array bot .. top of int;
```

Rewriting uses the rule botletop to prove that vec is nonempty.

```
rewrite;
```

A variable a of type vec is introduced. Such a declaration has no proof obligation.

```
var a : vec;
```

A predicate sort (for sorted) is defined on arrays. sort holds if the array is sorted in ascending order.

$sort(a) \triangleq \text{bot} \leqslant i \leqslant j \leqslant \text{top} \rightarrow a_j \geqslant a_i$

```
function sort(a) : bool =
begin
  all i, j : implies(and(int'ge(i, bot),
                          int'ge(j, i),
                          int'ge(top, j)),
                     int'ge(vec'extract(a, j),
                            vec'extract(a, i)))
end sort;
```

A predicate cont (for contains) is defined on an array, two indices and a target. cont holds when the array contains the target between the two indices.

$cont(a,i,j,t) \triangleq t \in a_i \ldots a_j$

```
function cont(a, i, j, t) : bool =
pre and(int'ge(i, bot),
        int'ge(j, i),
        int'ge(top, j))
begin
  some k : and(int'ge(k, i),
               int'ge(j, k),
               vec'extract(a, k) = t)
end cont;
```

A rule is defined which states that asking if a target is contained in an array between an index and itself is the same as asking if the array at that index is the target.

$$\frac{\text{bot} \leqslant i \leqslant \text{top}}{cont(a,i,i,t) \Leftrightarrow a_i = t}$$

```
axiom contii1(a, i, t) =
rule
begin
   implies(and(int'ge(i, bot),
               int'ge(top, i)),
           cont(a, i, i, t) = (vec'extract(a, i) = t))
end contii1;
```

The proof of this rule follows directly from the definition of `cont`.

```
invoke cont;
simplify without case analysis;
```

Another rule is introduced which states that, if an array at a given index is the target, then the array contains the target between the index and itself.

$$\frac{\text{bot} \leqslant i \leqslant \text{top} \ , \ a_i = t}{cont(a,i,i,t)}$$

```
axiom contii2(a, i, t) =
frule
triggers (vec'extract(a, i) = t)
begin
   implies(and(int'ge(i, bot),
               int'ge(top, i),
               (vec'extract(a, i) = t)),
           cont(a, i, i, t))
end contii2;
```

This rule's proof follows from the definition of `cont`. Note that it is necessary to highlight that there does exist a `k` between `i` and `i`, namely `i` itself.

```
invoke cont;
simplify without instantiation;
instantiate k = i;
simplify without instantiation;
```

Similarly the next rule infers that if the array at a given index is the target then the array contains the target between any two indices enclosing the target.

$$\frac{\text{bot} \leqslant i \leqslant k \leqslant j \leqslant \text{top} \ , \ a_k = t}{cont(a, i, j, t)}$$

```
axiom contmem(a, k, t) =
frule
triggers (vec'extract(a, k) = t)
begin
  all i, j : implies(and(int'ge(i, bot),
                         int'ge(k, i),
                         int'ge(j, k),
                         int'ge(top, j),
                         vec'extract(a, k) = t),
                     cont(a, i, j, t))
end contmem;
```

The proof of `contmem` follows directly from `cont`'s definition.

```
invoke cont;
simplify without case analysis;
```

This rule states that if there are two adjacent sections of the array, and the target is not in the lower section, then asking if the target is in the upper section is the same as asking if it is in either section.

$$\frac{\text{bot} \leqslant i \leqslant k < j \leqslant \text{top}\ ,\ \neg cont(a,i,k,t)}{cont(a,k+1,j,t) \Leftrightarrow cont(a,i,j,t)}$$

```
axiom contcont(a, i, k, j, t) =
rule
begin
   implies(and(int'ge(i, bot),
               int'ge(k, i),
               int'gt(j, k),
               int'ge(top, j),
               not(cont(a, i, k, t))),
           cont(a, plus(k, 1), j, t) = cont(a, i, j, t))
end contcont;
```

The proof of `contcont` proceeds from `cont`'s definition by partitioning `cont(a, i, j, t)` into `cont(a, i, k, t)` and `cont(a, plus(k, 1), j, t)`.

```
invoke cont;
split int'ge(k, k'2);
simplify without case analysis;
```

The following two rules about `cont` are for sorted arrays. These rules allow the size of the section of array being tested by a `cont` predicate to be reduced by eliminating portions which can not contain the target because their elements are too great or too small.

$$\frac{sort(a)\ ,\ \text{bot} \leqslant i \leqslant k \leqslant j \leqslant \text{top}\ ,\ a_k \geqslant t}{cont(a,i,j,t) \Leftrightarrow cont(a,i,k,t)}$$

```
axiom contpart1(a, k, t) =
frule
triggers (int'ge(vec'extract(a, k), t))
begin
  all i, j : implies(and(sort(a),
                         int'ge(i, bot),
                         int'ge(k, i),
                         int'ge(j, k),
                         int'ge(top, j),
                         int'ge(vec'extract(a, k), t)),
                     cont(a, i, j, t) = cont(a, i, k, t))
```

`contpart1` is commenced by eliminating the case where `k` is the target and the case where the array does not contain the target. The proof is completed by using instantiation on the `sort` predicate.

```
split vec'extract (a, k) = t;
rewrite;
split cont (a, i, j, t);
reduce;
invoke cont;
prenex;
instantiate i'0 = k, j'0 = k'0;
rewrite;
```

$$\frac{sort(a)\ ,\ \text{bot} \leqslant i \leqslant k \leqslant j \leqslant \text{top}\ ,\ a_k \leqslant t}{cont(a,i,j,t) \Leftrightarrow cont(a,k,j,t)}$$

```
axiom contpart2(a, k, t) =
frule
triggers (int'ge(vec'extract(a, k), t))
begin
  all i, j : implies(and(sort(a),
                         int'ge(i, bot),
                         int'ge(k, i),
                         int'ge(j, k),
                         int'ge(top, j),
                         int'ge(t, vec'extract(a, k))),
                     cont(a, i, j, t) = cont(a, k, j, t))
end contpart2;
```

The proof of `contpart2` follows the same pattern as the proof of `contpart1`.

```
split vec'extract (a, k) = t;
rewrite;
split cont (a, i, j, t);
reduce;
invoke cont;
prenex;
instantiate i'0 = k, j'0 = k'0;
rewrite;
equality substitute t;
rewrite;
```

This last rule states that if the array contains the target between two indices then, the array will contain the target between any two enclosing indices.

$$\frac{\text{bot} \leqslant k \leqslant i \leqslant j \leqslant l \leqslant \text{top}\ ,\ cont(a,i,j,t)}{cont(a,k,l,t)}$$

```
axiom contexp(a, i, j, t) =
frule
triggers (cont(a, i, j, t))
begin
  all k, l : implies(and(int'ge(k, bot),
                         int'ge(i, k),
                         int'ge(j, i),
                         int'ge(l, j),
                         int'ge(top, l),
                         cont(a, i, j, t)),
                     cont(a, k, l, t))
end contexp;
```

The proof of `contexp` proceeds directly from the definition of `cont`.

```
invoke cont;
rewrite;
```

A boolean variable `found` is introduced. It will be used by the search procedure to signal the presence of the target in the array being searched.

```
var found : bool;
```

Below is the searching procedure itself. `search` searches the array `a` for the target `t`. `found` is set to true if and only if `a` contains `t`, in which case `i` is set of the index at which `a` is `t`.

```
procedure search(a, t, found, i)
pre sort(a)
post a' = a ∧
       t' = t ∧
       found' = cont(a, bot, top, t) ∧
       found' ⇒ bot ⩽ i' ⩽ top ∧ a_i' ⩽ t
begin
  i, j := top, bot
  while a_i ≠ t ∧ i ≠ j do
   { sort(a) ∧ bot ⩽ i ⩽ j ⩽ top ∧ cont(a, bot, top, t) ⇔ cont(a, i, j, t) }
    m := mid(i, j)
    if a_m = t then
      i := m
    elseif a_m < t then
      i := m + 1
    else { a_m > t }
      j := m
    end if
  end while
found := a_i = t
end search
```

```
procedure search(lvar a, lvar t, pvar found, pvar i) =
pre sort(a)
post and(found = cont(a, bot, top, t),
         implies(found, and(int'ge(i, bot),
                            int'ge(top, i),
                            vec'extract(a, i) = t)))
begin
  pvar j
  pvar m

  i := bot
  j := top
  loop
    invariant and(sort(a),
                  int'ge(i, bot),
                  int'ge(j , i),
                  int'ge(top, j),
                  cont(a, bot, top, t) = cont(a, i, j, t))
    measure ordinal'val(minus(j, i))
    exit when or(vec'extract(a, i) = t, i = j)
    m := mid(i, j)
    if vec'extract(a, m) = t then
      i := m
    elseif int'lt(vec'extract(a, m), t) then
      i := plus(m, 1)
    else
      j := m
    end if
  end loop
  found := vec'extract(a, i) = t
end search;
```

The proof of correctness of the procedure `search` can now be commenced. The proof is approached by first attacking all the trivial subexpressions relating to the ordering of `i` and `j`, and then using the axioms declared above to discharge the more complex portions of the proof. Below is given a step-by-step description of the proof (a full transcript of the proof session, with machine responses, is not given because it is much too long):

First some general simplifications are made, making use of the fact (from the loop invariant) that `cont(a'0, bot, top, t) = cont(a'0, i, j, t)`:

```
simplify without instantiation;
equality substitute cont(a'0, bot, top, t);
```

Next `contmem` is used to eliminate the case where `vec'extract(a, i) = t`, and `contii1` to eliminate the case where `i = j`.

```
equality substitute j;
rewrite;
```

In the next set of proof steps midincl is used to clean up expressions like int'le(i, mid(i, j)) and int'le(bot, mid(i, j)):

```
split int'ge(mid(i, j), i);
simplify without instantiation;
split int'ge(j, i);
rewrite;
```

At this point midinc2 is used to clean up expressions like int'lt(mid(i, j), j) and int'le(mid(i, j), top):

```
split j <> mid (i, j);
split int'ge (j, mid (i, j));
simplify without instantiation;
split int'ge (j, i);
rewrite;
```

Now contmem is used to eliminate the case where vec'extract(a, mid(i, j)) = t, and contpart1 to eliminate the case where int'ge(t, vec'extract(a, mid(i, j))):

```
split and(int'ge(mid(i, j), i),
          int'ge(j, mid(i, j)));
rewrite;
```

The cases introduced by the previous split command are cleaned up as follows:

```
split int'ge(j, i);
rewrite;
```

Now the formula is placed in a form suitable for use with instantiated facts:

```
prenex;
open;
```

At this point the information in the rule contcont is introduced, instantiated as above, and the disable command is used to prevent rewrite from simplifying the information away.

```
use contcont a = a'0, i = mid(i, j), k = mid(i, j),
             j = j, t = t;
disable contcont;
```

The following proof steps rearrange the formula so that rewrite uses information just introduced to simplify the case where i = m + 1, and replace cont(a'0, plus(mid(i, j), 1), j, t) by cont(a'0, mid(i, j), j, t).

```
split int'ge(mid(i, j), i);
simplify without instantiation;
split int'ge(j, i);
rewrite;
split j = mid(i, j);
rewrite;
split int'ge(j, mid(i, j));
split int'ge(j, i);
rewrite;
equality substitute cont(a'0, plus(mid(i, j), 1), j, t);
```

Exploiting the fact that, since all the other cases have now been simplified away, it is known that `int'le(t, vec'extract(a'0, mid(i, j)))`:

```
rearrange;
split int'le(t, vec'extract(a'0, mid(i, j)));
rewrite;
```

The information in the rule `contpart2` is now introduced, instantiated as above, and again `rewrite` is prevented from simplifying the information away.

```
use contpart2 a = a'0, k = mid(i, j), t = t;
disable contpart2;
```

Next, instantiate the quantified variables introduced from `contpart2` for use in this formula:

```
instantiate i'0 = i, j'0 = j;
```

The formula is now rearranged to highlight the fact that each of the hypotheses of the introduced rule is known:

```
split and(int'ge(i, bot),
          int'ge(top, j),
          int'ge(j, i),
          sort(a'0),
          int'ge(t, vec'extract(a'0, mid(i, j))));
```

Finally, the introduced rule is used to simplify the last case where `int'lt(t, vec'extract(a, mid(i, j)))`, thus completing the proof:

```
rewrite;
```

Department of Defence

DOCUMENT CONTROL DATA SHEET

Page Classification	
UNCLASSIFIED	
Privacy Marking/Caveat	
N/A	

1a. AR Number	1b. Establishment Number	2. Document Date	3. Task Number
AR-006-482	ERL-0545-RR	MARCH 1991	DST 90/175

4. Title	5. Security Classification	6. No. of Pages	44
REPORT ON m-EVES	U (Document) U (Title) U (Abstract)	7. No. of Refs.	18

S (Secret) C (Confi) R (Rest) U (Unclass)

* For UNCLASSIFIED docs with a secondary distribution LIMITATION, use (L)

8. Author(s)	9. Downgrading/Delimiting Instructions
J. Grundy	N/A

10a. Corporate Author and Address	11. Officer/Position responsible for
Electronics Research Laboratory PO Box 1600 SALISBURY SA 5108	Security.....N/A..... Downgrading.....N/A..... Approval for Release.....D/ERL.....
10b. Task Sponsor -	

12. Secondary Release of this Document

APPROVED FOR PUBLIC RELEASE

Any enquiries outside stated limitations should be referred through DSTIC, Defence Information Services, Department of Defence, Anzac Park West, Canberra, ACT 2600.

13a. Deliberate Announcement

NO LIMITATION

13b. Casual Announcement (for citation in other documents)

[√] No Limitation

[ ] Ref. by Author & Doc No only

14. DEFTEST Descriptors	15. DISCAT Subject Codes
*m-Eves (Environment for verification and evaluation software) Software evaluation Computer program verification *m-Verdi (Programming Language) *m-NEVER (theorem prover)	1205

16. Abstract

This report reviews the verification tool m-EVES (Environment for Verifying and Evaluating Software). It describes the two main components of m-EVES: the language, m-Verdi and the theorem prover, m-NEVER. The main purpose of this report is to describe how problems are attacked in m-EVES, and draw conclusions about the strengths and weaknesses of m-Verdi and m-NEVER. Where appropriate, comparisons are made with other program verification and theorem proving environments.

Page Classification
UNCLASSIFIED
Privacy Marking/Caveat
N/A

16. Abstract (CONT.)

17. Imprint

Electronics Research Laboratory
PO Box 1600
SALISBURY SA 5108

18. Document Series and Number	19. Cost Code	20. Type of Report and Period Covered
ERL-0545-RR	800439	ERL RESEARCH REPORT

21. Computer Programs Used

m-EVES

22. Establishment File Reference(s)

N/A

23. Additional information (if required)